IT UNIVERSITY OF COPENHAGEN

DevOps, Software Evolution and Software Maintenance

COURSE CODE: KSDSESM1KU

MSc IN COMPUTER SCIENCE

DevOps: ITU-MiniTwit

GROUP E — GRL PWR

IT UNIVERSITY OF COPENHAGEN

Name	**Email**
Amalie Bøgild Sørensen	abso@itu.dk
Andreas Nicolaj Tietgen	anti@itu.dk
Malin Birgitta Linnea Nordqvist	bino@itu.dk
Mille Mei Zhen Loo	milo@itu.dk
My Marie Nordal Jensen	myje@itu.dk
Sarah Cecilie Chytræus Christiansen	sacc@itu.dk

13th May 2024

2500 words

Part

Table of Contents

1 System's Perspective

1.1 Design and Architecture

Figure 1 provides an overview of our system infrastructure:

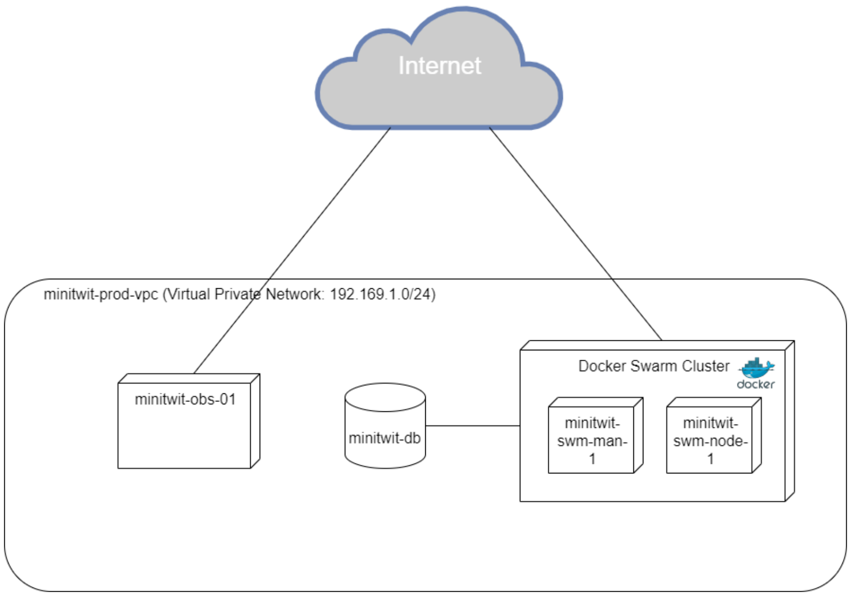


Figure 1: *Overview of the system*

The system includes a single Virtual Private Network with an IP 192.168.1.0/24. Within this network, we operate a database cluster consisting of a single node. This node is connected to the Docker Swarm cluster, featuring both a manager and a worker node.

Both our monitoring and cluster servers are visibly exposed to the internet. Additionally, the database server is accessible from the internet but has constraints to only talk to our servers and therefore not accept requests from any other host.

1.2 Interactions of Subsystems

Our system consists of two main subsystems, namely the REST SimulatorAPI and the webapplication Minitwit. As can be seen in figure 2, The SimulatorAPI and Minitwit subsystems are similar in structure. However when looking at the processes behind the structures in Figure 3 and 4 it is apparant that these two differ a great deal.

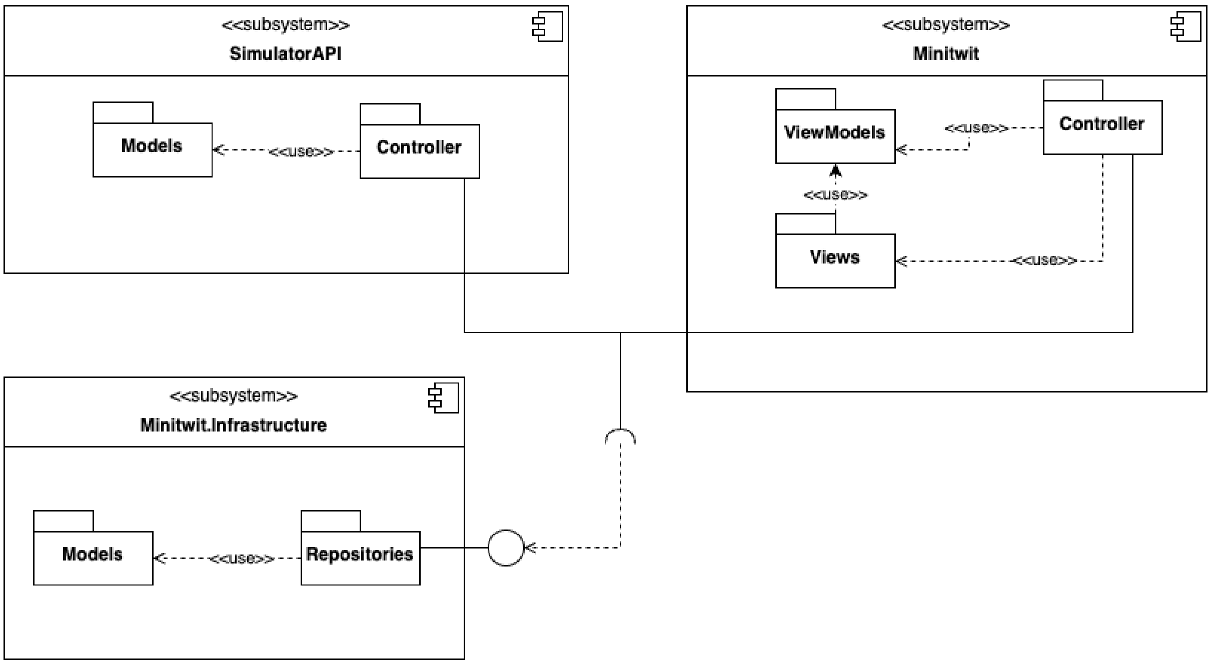


Figure 2: *An overview of the subsystems and how they interact*

Because the two main subsystems are inherently different we wanted to implement a database abstraction layer such that we had a uniform way of querying the database. Therefore, both controllers use the Repository package from the subsystem Minitwit.Infrastructure to query the database using the Repository pattern[5]. Due to the different specifications for the Minitwit and SimulatorAPI, the controller for Minitwit is more complex. As can be seen in figure 3 the controller for Minitwit has to present information in a GUI, whereas the SimulatorAPI only calls the repository to fetch the user messages, as can be seen in figure 4.

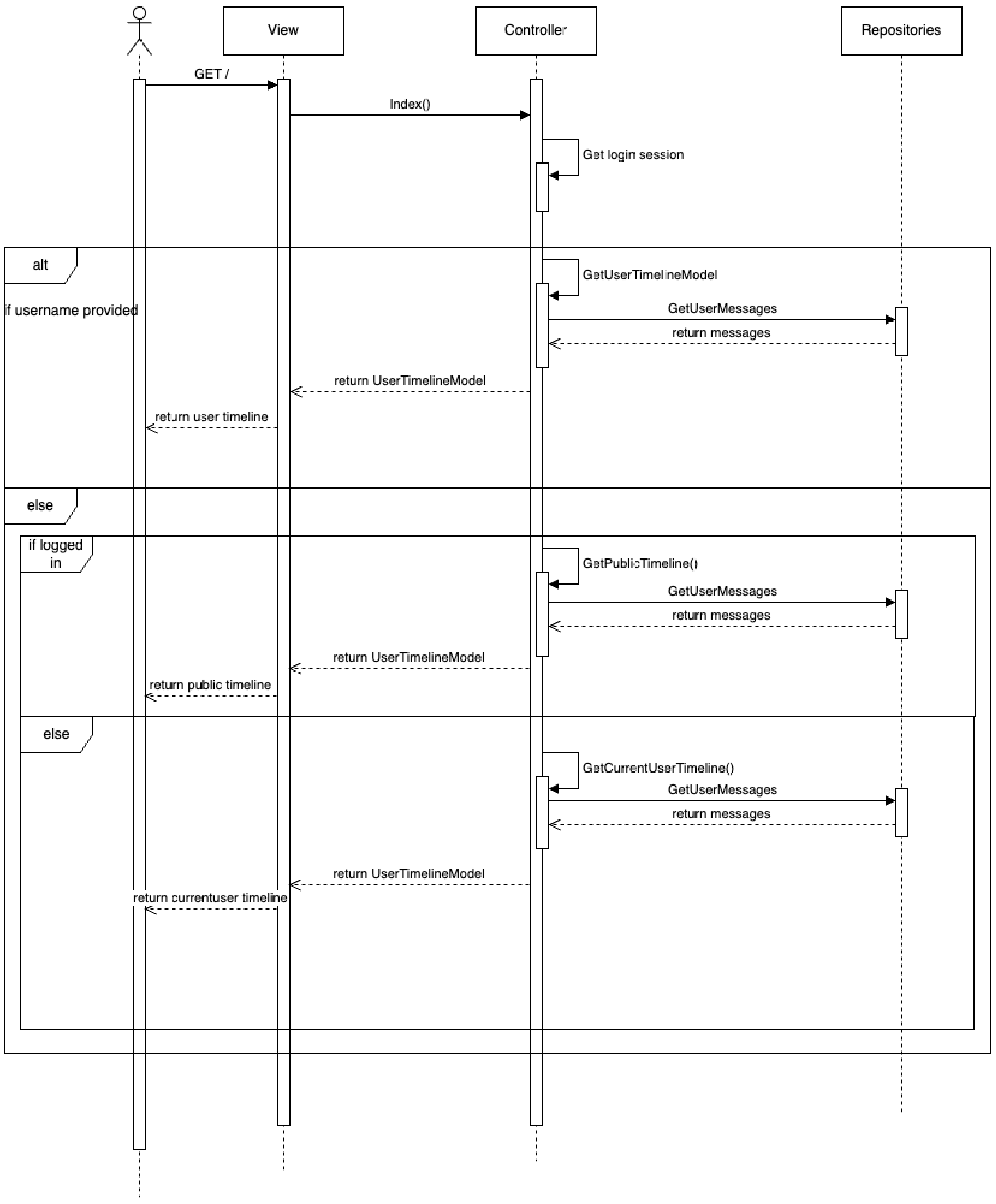


Figure 3: *Minitwit: Sequence diagram of a user call to the index page*

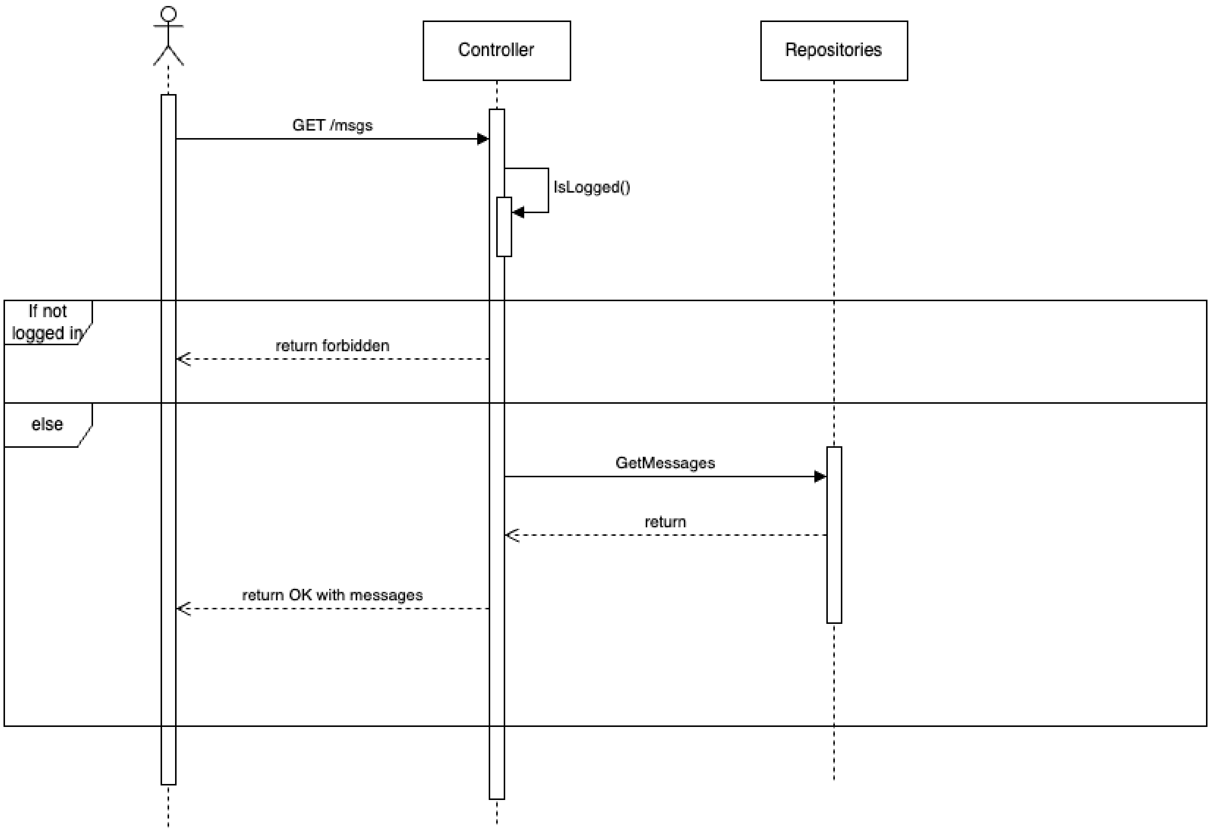


Figure 4: *SimulatorAPI: Sequence diagram of a user call to get messages*

1.3 Current state

Table 1 describes the status of the system after the simulator has been shut down.

Total requests	34233
Requests secured	34233
Total unhandled exceptions	18
P99 min response time	94.2 ms
P99 max response time	716 ms
Number of users	116995
Avg followers per user	28.5

Table 1: *System metrics*

Furthermore, the static analysis tool Sonarcloud assesses that the quality of the overall code is good (see appendix B).

2 Process Perspective

2.1 CI/CD Chain

In order to introduce continuous integration and continuous deployment to the program, we used Github Actions, namely workflows. The project included several different workflows like provision and deploy as described in table 2.

Workflow file	**Function**
AutomaticBuildRelease.Backend	Bumps the patch versioning of the *MinitwitSimulatorAPI* project and creates a release on the github repository.
AutomaticBuildRelease.Frontend	Bumps the patch versioning of the *Minitwit* project and creates a release on the github repository.
WeeklyMinorRelease	Bumps the minor version of the entire program and creates a release each Thursday at 20:00 UTC.
Linters-and_Formatter	Runs Hadolint, Pre-commit, and the dotnet-format linter, when a PR is created, modified, and reopened.
Testing	Runs all tests on the infrastructure, the frontend, and the backend, when a PR is created, modified, and reopened.
Provision-and-Deploy	Provision and deploy swarm and observability servers using Ansible for server provision and Pulumi for infrastructure as code, on pushes to main.
latex-build	Creates a report PDF when changes have been made to the report.

Table 2: *The function of the workflows*

Figure 5 vizualizes the development pipeline all the way from writing code to production. This includes running precommit to check for formatting to passing both UI and integrations tests as well as building and deploying.

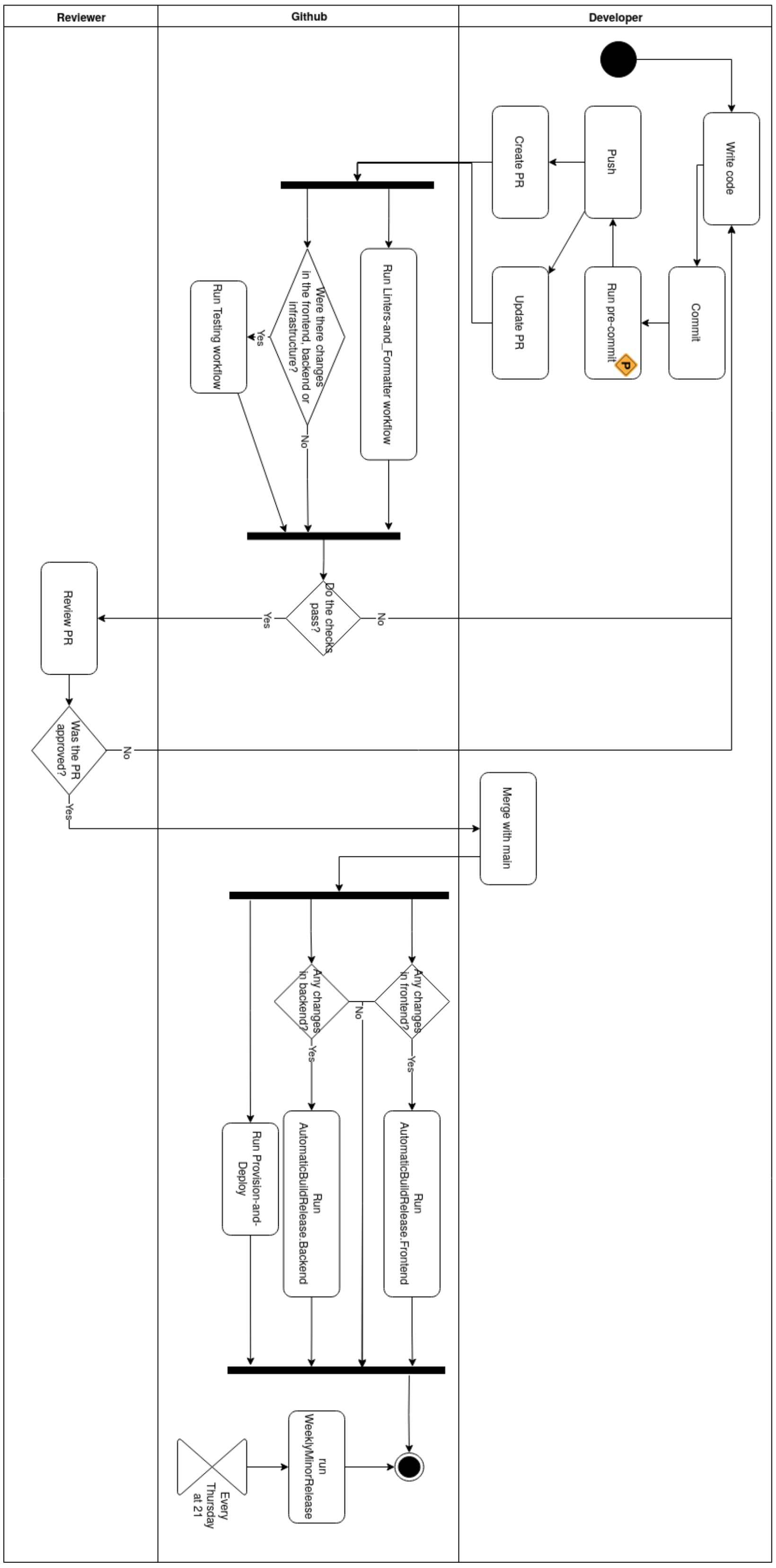


Figure 5: *Activity diagram showcasing the workflows interacting*

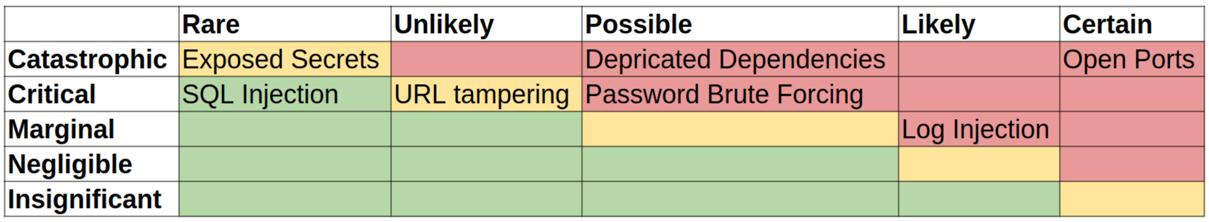
2.2 Monitoring and logging

For monitoring we use Grafana. We export metrics by using OpenTelemetry and collect the metrics via Prometheus and lastly push them to Grafana. Our board shows requests duration, errors rate, top 10 unhandled exception endpoints and more (for visualization of the board, see appendix A). We have used the board to get an overview of where to put our focus, ie. which endpoints to improve, which errors to fix, etc. Moreover, we are gaining insights into the health of the system and get an impression of how the backend and frontend handle the requests.

For logging, we use Grafana and Loki. It seemed natural to continue our work with Grafana in order to keep the system setup as simple as possible. The logs are divided into Information, Warning, Debug and Error. All logging statements are placed in the controllers, such that we have information about the users' whereabouts in the system. For example, we log when a user logins whether successful or not.

2.3 Security Assessment

Based on our security assessment (see appendix E), we constructed the following risk matrix:



	Rare	**Unlikely**	**Possible**	**Likely**	**Certain**
Catastrophic	Exposed Secrets		Depricated Dependencies		Open Ports
Critical	SQL Injection	URL tampering	Password Brute Forcing		
Marginal				Log Injection	
Negligible					
Insignificant					

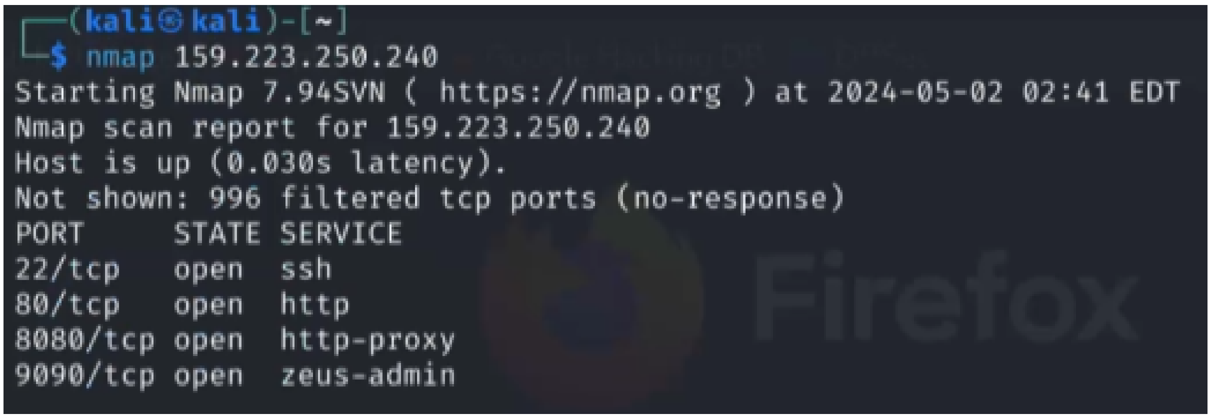
Figure 6: *Risk matrix from security assessment*

We decided to focus on the scenarios in the red zone, as they would be the cause of most damage. Due to the team not having enough knowledge about how the simulator worked in regards to creating users and logging in, we did not move further with the *Password Brute Forcing* scenario. However, possible solutions could be to incorporate 2FA, password requirements and time-out limitation for an actual system in production.

For *Log Injection*, we made sure to sanitize user inputs, as they previously were put directly into the log, which could be exploited to hide ill-intentioned actions. We thereby hardened the system against injection attacks.

With *Depricated Dependencies*, we chose to integrate the tool *dependabot* into our CI/CD pipeline to watch out for outdated dependencies in our main repository. This helps us ensure that an adversary is not able to take advantage of the team not being aware of vulnerabilities in variuos of the used tools.

Open Ports was deemed the biggest risk, as this was most likely the reason for us being hacked, which we have written about in further details in section 3.1. We used the tool *nmap* to scan the IP address of our application and check for open ports, which showed the following:



```
┌──(kali㉿kali)-[~]
└─$ nmap 159.223.250.240
Starting Nmap 7.94SVN ( https://nmap.org ) at 2024-05-02 02:41 EDT
Nmap scan report for 159.223.250.240
Host is up (0.030s latency).
Not shown: 996 filtered tcp ports (no-response)
PORT     STATE SERVICE
22/tcp   open  ssh
80/tcp   open  http
8080/tcp open  http-proxy
9090/tcp open  zeus-admin
```

Figure 7: *Open ports found by nmap*

We tried to close port 9090, which was used for Prometheus, but it was unclear if we were successful as *nmap* kept showing the port as open, despite seeing the firewall configuration say otherwise on the server itself. This was likely caused by Docker bypassing our firewall, which was handled by the Ansible ufw module, through IP tables.

2.4 Scaling

We use horizontal scaling, as we wanted a more resilient application and limit downtime when deploying. We settled on Docker Swarm over building a custom load balancer or adopting Kubernetes, primarily due to its simplicity and compatibility with the existing system setup. To implement Docker swarm we created a new server in our system and added a manager and a worker node to the swarm.

When attempting to scale by adding another frontend replica, we encountered challenges. In order for the two replicas to work together we needed to handle distributed sessions to remember who is logged in even though the frontend replica is switched. However, we did not succeed in this (we refer to week 11 in the log in appendix D), which is the reason why we only have a single frontend replica.

For the update strategy we went with rolling updates because it is the default update strategy for Docker Swarm.

3 Lessons Learned

3.1 Lesson 1: Getting Hacked

Just a couple of hours after attending the lecture on security, we got hacked, leading us to experience firsthand how important it is to incorporate security in a CI/CD pipeline.

The first suspicion we got was when we discovered, through our monitoring, that the response time of our server was suddenly very slow. This led us to our Digital Ocean dashboard which showed that the server was using 100% CPU power, which is highly unusual.

From that point the group scoured the server for clues as to what was happening, finding countless calls to `masscan` essentially drowning our server, as well as mysterious installations and what looked like a call to a remote script via a cronjob.

After a few failed attempts to evict the adversary it was decided to destroy the server, as we fortunately had already implemented our Infrastructure as Code, so provisioning and deploying a new server could be done in under half an hour.

After some introspection into our system, we assume that the adversary had gotten access via some open ports that we were unaware were exposed. The ports became exposed in an attempt to make the network function between servers in a docker swarm, which seems to override the firewall.

Learning from this, we have worked to close exposed ports from Docker and finding alternative solutions to setting up the network. Another key takeaway is that because we had the necessary monitoring in place, to figure out that the server was being targeted, as well as having implemented Infrastructure as Code, we were able to detect and react to the attack fairly quickly, giving us only a few hours of downtime.

3.2 Lesson 2: Shift from Vagrant to Ansible-Pulumi

At the beginning of the project, we had chosen to provision our VMs with Vagrant, inspired by the exercises from the course. We realized later that we would have to switch Digital Ocean account at some point due to running out of credit, which then meant we had to streamline the setup of our VMs.

The choice of Configuration Management tool fell upon Ansible, which was supposed to be called by Vagrant in a config server, provisioning the web and monitoring servers. However, it turned out that Vagrant was not the right tool for the job when having more complicated automation and collaboration needs for our project. After many hours of attempting to get Vagrant to work with Ansible, we found out that Vagrant saves local metadata to maintain some state, which was making the provisioning from Ansible and the config server fail[2]. Upon further research we also found out that the creators of Vagrant had created the tool for setting up development enviroments and not for maintaining production infrastructure[3]. Furthermore, Vagrant does not seem to be supported with GitHub Actions, which is likely due to how it handles state.

We decided to cut our losses with Vagrant and search for a more suitable tool that could help us write our infrastructure as code, where the choice fell upon Pulumi. This taught us the importance of thoroughly researching the available tools to navigate through their advantages and drawbacks. It was a valuable lesson to see the difference it made when taking the time to investigate different tools and their properties in order to make an informed decision based on the knowledge of our system's needs.

3.3 DevOps Style

When reflecting on how we as a group incorporated the style of DevOps into our way of working, we recalled the *Three Ways*, which were the characterising principles for processes and behaviour in DevOps, that consisted of *Flow*, *Feedback* and *Continual Learning and Experimentation*[4].

In regards to the principle of flow, it wasn't hard to adopt the ideas of making our work visible and reducing batch sizes, as all members of the group have previously worked agile in other courses, which also embodies those same ideas. Using a kanban board to track our tasks and their progress not only helped us stay on target, but also helped us with confining each task such that it could be deployed continuously[4].

For the principle of feedback, the concept of peer reviewing via pull requests on GitHub helped install a sense of ownership over the application. Automating the process of not only testing, but also building and deploying the entire application in a continous fashion also allowed for errors to be found and mitigated quickly compared to earlier projects we have worked on, where it was easy for the issues to pile up on each other.

Lastly, with the principle of continual learning and experimentation, having a safe system of work[4] was crucial for us to learn and grow in a secure environment, which in turn allowed for greater experimentation in how to improve the system and its setup. By keeping weekly work logs (see appendix D) and guides easily available, each member would have the same oppertunity to gain a deeper and better understanding of the system as a whole.

4 References

[1] Ansible. *Docker Swarm Module*. 2024. URL: https://docs.ansible.com/ansible/latest/collections/community/docker/docker_swarm_module.html.

[2] DevOps Group e. *Github Issue: Add playbooks in the vagrant provisioning steps*. 2024. URL: https://github.com/devops2024-group-e/itu-minitwit/issues/178.

[3] Hashicorp. *Vagrant vs. Terraform*. 2024. URL: https://developer.hashicorp.com/vagrant/intro/vs/terraform.

[4] Gene Kim. *The DevOPS Handbook - How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. It Revolution Press, 2016. ISBN: 9781942788003.

[5] Microsoft. *Design the infrastructure persistence layer: The Repository Pattern*. 2023. URL: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design#the-repository-pattern.

[6] OpenTofu. *The OpenTofu Manifesto*. 2024. URL: https://opentofu.org/manifesto/.

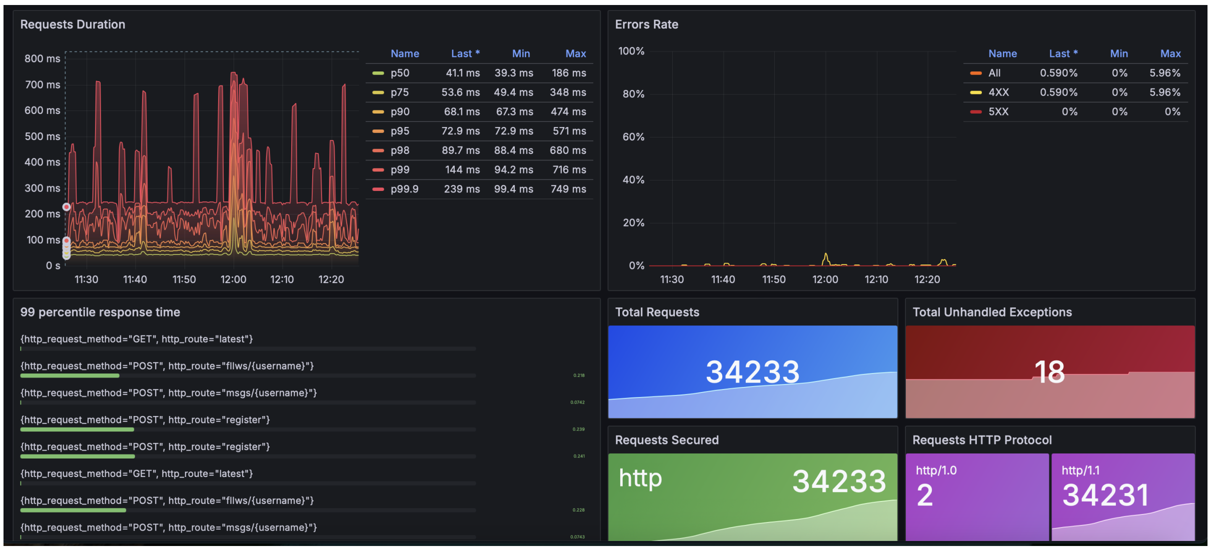
[7] OWASP. *OWASP Top Ten*. 2021. URL: https://owasp.org/www-project-top-ten/.

Part

Appendices

Table of Contents

A Grafana


Requests Duration

Name	Last *	Min	Max
p50	41.1 ms	39.3 ms	186 ms
p75	53.6 ms	49.4 ms	348 ms
p90	68.1 ms	67.3 ms	474 ms
p95	72.9 ms	72.9 ms	571 ms
p98	89.7 ms	88.4 ms	680 ms
p99	144 ms	94.2 ms	716 ms
p99.9	239 ms	99.4 ms	749 ms

Errors Rate

Name	Last *	Min	Max
All	0.590%	0%	5.96%
4XX	0.590%	0%	5.96%
5XX	0%	0%	0%

99 percentile response time

{http_request_method="GET", http_route="latest"}	
{http_request_method="POST", http_route="fllws/{username}"}	0.218
{http_request_method="POST", http_route="msgs/{username}"}	0.0742
{http_request_method="POST", http_route="register"}	0.239
{http_request_method="POST", http_route="register"}	0.241
{http_request_method="GET", http_route="latest"}	
{http_request_method="POST", http_route="fllws/{username}"}	0.228
{http_request_method="POST", http_route="msgs/{username}"}	0.0743

Total Requests: 34233

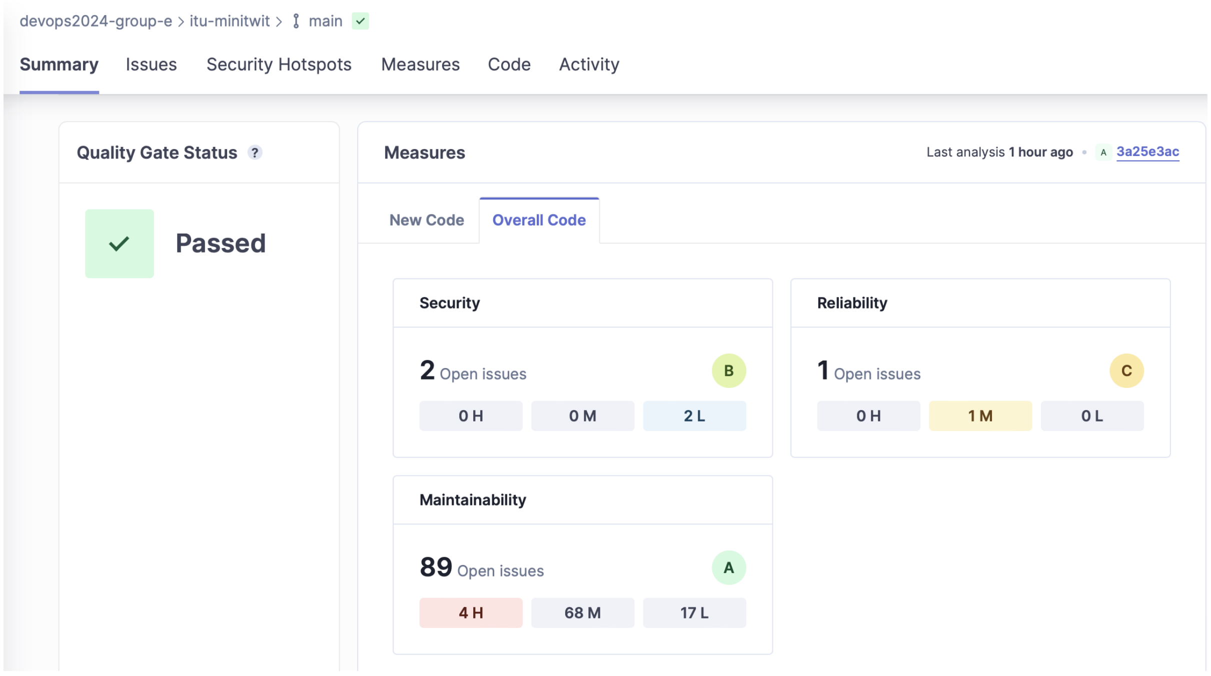
Total Unhandled Exceptions: 18

Requests Secured: http 34233

Requests HTTP Protocol: http/1.0 2; http/1.1 34231

Figure 9: *A snapshot of the Grafana dashboard.*

B Sonarcloud



Quality Gate Status

Passed

Measures

Last analysis 1 hour ago · A 3a25e3ac

New Code | Overall Code

Security: 2 Open issues, B. 0 H, 0 M, 2 L

Reliability: 1 Open issues, C. 0 H, 1 M, 0 L

Maintainability: 89 Open issues, A. 4 H, 68 M, 17 L

Figure 10: *Sonarcloud code analysis.*

C Decision Log

C.1 Language: C#

We decided to use C# for refactorization. We did this due to C# being object-oriented and a stable language. C# is used by many companies and is therefore known to many. Because C# is used by many there also exists good documentation easily accessible. There exist both micro web-frameworks and more advanced front-end frameworks for C# in case we want to develop the application further. C# is also a part of the .NET platform, which allows for language interoperability and access to the Core framework used for web applications.

C.2 Micro web-framework: Razor Pages

Initially, we decided to work with Scriban as the micro web-framework, as opposed to e.g. Razor Pages, because the refactorization from the original version of Minitwit seemed more direct in Scriban. This is due to the fact that Scriban works with `.html` templates, like we have in the original version, whereas Razor Pages integrates C# in HTML, which would lead to more "merging" of code. Furthermore, Scriban's parser and render are both faster and use less memory than other templates. After trying to find information on Scriban, the group had trouble figuring out how to set it up properly. Comparing the amount of documentation for Scriban versus Razor Pages, we decided to switch the micro web-framework to Razor Pages, as this would allow us to start working with the material faster. Razor Pages also has simple context of structure and is flexible to fit with any application. Another reason for choosing Razor Pages is that each page is self-contained, with the view and code organized together. This makes Razor Pages more organized and easier to work with. With the documentation, flexibility and the organization, Razor Pages felt like a good choice for a micro web-framework.

C.3 Work log: Notion

To start of with, we decided to keep our work log/diary in GitHub, since this automatically provides us with useful information, e.g. dates and authors. Additionally, the work log will follow the code completely, and we can see which code belongs with what log entries. However, we realized after a few weeks that this did not work as intended. The reason was that the action of making a pull request and requesting a review after just a tiny addition to the log seemed too complicated. As a consequence, we ended up keeping our log in Notion and then moving it to GitHub afterwards, which was double the work and definitely not the intention from the start. Therefore, we decided to properly move the log to Notion, which also has the advantage that everyone can follow each other's writing in real time and collaborate in writing simultaneously.

C.4 Database connection: Entity Framework

We have chosen Entity Framework (EF) for our database connection as it works well with .NET. EF has the possibility of generating a model from an already existing database and lets us load data using C# classes. Additionally, EF supports LINQ and works well with SQLite.

C.5 Server host: Digital Ocean

We have chosen Digital Ocean as our cloud hosting service, as it is a cheap cloud hosting service that allows for developers to have quite a lot of control over their server hosting. Furthermore, popular cloud hosting alternatives have a tendency to be more expensive, however typically not by a lot. Even though this increased expense often comes with a lot of additional configuration options, some of the students in the group have limited experience working with cloud servers, wherefore we believed Digital Ocean would be a good choice, as it is free, and there are plenty of guides and good documentation.

C.6 CI/CD: GitHub Actions

For CI/CD we chose GitHub Actions. The reason is that we are already using the GitHub platform and that it is well documented. Furthermore, it is easy to configure secrets and refer to them from the `.yaml` file. Additionally, it collaborates nicely with Digital Ocean which hosts our servers.

C.7 Database: SQLite to PostgreSQL

We chose to use PostgreSQL when we had to start creating our Dockerfile. We looked for a Microsoft or another official Docker image from a good source (that is from a big, known company, that we can trust). But we could not find one and we saw that Helge used something he created himself. Thus, we had to choose among some of the databases that has been containerized. That were the first two requirements. The third requirement is that we need a relational database. This excluded well-known document databases like MongoDB. At the end we had two good options. Microsoft SQL and PostgreSQL.

Feature-wise they offer the same things. They are both good at handling concurrent requests and have transactions. They both offer the possibility to add indexes, and to have multiple servers act as a cluster in order to handle even more requests.

We chose the solution that we have had experience with before. Both in our previous course *Introduction to Database Systems*, and in previous projects done at ITU. We have had an abundance of new technologies throughout our times as students, so it felt almost relieving to not use a new one again.

C.8 Monitoring: Grafana

Looking for a tool to visualize the metrics of dotnet applications, and maybe other tools in our tech stack as well, required a versatile tool. That is, we needed to have a tool where we could search and visualize the current status of our application. Obviously, a lot of tools can handle this. For that reason it is nice to see that the tool OpenTelemetry is announced and widely used - both as a collector and exporter.

Thus, a requirement is that the monitoring system has to be able to integrate with OpenTelemetry in order to make it easier to change monitoring system if required.

Furthermore, it would be prefered to have an open source and free of charge product that is being maintained.

C.9 Metrics collector: Prometheus

There has not been an elaborate thought process about the choice of Prometheus. It is open source and can collaborate with the OpenTelemetry exporter. Additionally, it is able to connect to the Docker metrics as well.

C.10 Database migration: Digital Ocean

We have considered hosting a separate server on Digital Ocean, and using Digital Ocean's own database service. We considered Digital Ocean's options, due to the wide array of databases that they support. However, when researching about troubleshooting, and problems using this solution, we found very few answers from crowd sourced platforms such as StackOverflow, and a lot of guides from the Digital Ocean docs themselves.

C.11 Logging stack: Grafana + Loki + Promtail

We believe that the amount of applications used can get very bloated. Therefore, we have decided to use Grafana-Loki, as we have already setup Grafana last week when implementing monitoring.

C.12 Configuration management: Ansible

We realise now that we need a tool that easily manages the desired state of our running VMs. Especially now that we are going to create more VMs and that we have to move our infrastructure before the 19th of April to another Digital Ocean account.

For that reason we need a simple way to have a configuration of our VMs defined that manages our provisioning. Our requirements are the following:

- Simple setup
- Should be pushable i.e. as soon as we have changed the configuration, then it should push the desired state out to our servers
- Keeps track if specific portions of the provisioning scripts already have been run
- Does not require us to learn a new language

We have been looking at different players in the market. Especially we have been looking at the CNCF and asking colleagues. Here are some of the products we have looked at:

- Ansible
- Fabric - Not really a configuration management system but a “simple” push-based script system
- Chef

C.12.1 Fabric

Fabric seemed nice, however it lacked some of the basic things that we wanted to solve, such as the idempotence of the desired state of the VM. This we would have to script ourselves. Also we would have to add another language to our repo, that is Python, which would require us to setup our development container again to have both dotnet and Python. Our experience with that was that it was very hard to get right, and we finally have a setup now that works on all of our laptops. So we did not want to break that.

C.12.2 Chef

Chef is a configuration management system, just like Ansible as well. It provides a hybrid solution of pull- and push-based. That is, we push the configuration management changes to a central server and then the VMs will pull the changes from the server (probably after some interval).

As can be seen this also requires that the VMs has the Chef Client installed and configured to point at the Chef Server. This provides the nice feature that the Chef Server do not have to know which servers to configure. It is up to the Chef Clients to know where to pull the configurations from.

The downside of this setup is the complexity. It seems like a lot of things that needs to be taken care of, and the fact that we would have a load overhead by adding clients to our small web server VMs would maybe

In order to create a configuration we would have to create what they call cookbooks. This is written in Ruby as far as we can see. The nice thing about this is that there are testing frameworks implemented, which provides a way to be sure that the configuration works and configures as we intend it. However, while experimenting with Vagrant it has become obvious that our group does not have sufficient knowledge in Ruby and we would currently rather invest time in becoming better at using the observability tools that we have, rather than learning a new language.

From a security point of view Chef uses some of the protocols that we have already opened ports for. That is, it uses standard HTTP protocol to communicate between the server and nodes, and authorizes through certificates.

C.12.3 Ansible

Some of the nice features in Ansible is that the setup seems fairly simple:

- One server to distribute to a list of servers
- No additional clients to install
- No code for defining configurations, it uses a YAML to define the configuration of the VMs

The downside is that there is a central server that needs to have a list of the created VMs, which then needs to be maintained. Furthermore, from a security perspective, we would have to allow `ssh` and that the Ansible server would have to store the private `ssh` key to access all of the servers. We have a hard time trying to understand how we are supposed to push this to the Ansible server and then run it? Would that be another `ssh` session?

Looking a bit further into it is seems like the way to automate the configurations of the server can be done with a proprietary product called Ansible Automation, which requires a license. But some has been able to create a custom GitHub Action that can push these changes to the servers. This requires that the `ssh` port is open to the public, which it already is.

As with Chef, Ansible also has their own term for a configuration, i.e. a playbook. The term is from American sports, where each task in the playbook is a play that constitutes the playbook. Each play in the playbook is idempotent. Thus, it seems to provide simple idempotence, which is fine in most cases, but it some cases it may require some more advanced idempotence.

C.12.4 Why do we choose Ansible?

We chose Ansible because the setup seems simple. It provides the features that we need i.e. an easy setup that does not require a new client to be installed on our application servers. Configurations described in YAML and not in some other language like Ruby or Python.

C.13 Adding E2E and integration tests early

Early in the project process we were given some integrations and end-to-end (E2E) tests for the frontend and Simulator API, respectively. This was beneficial when we had to develop our system to begin with, but we quickly realised that in order to be comfortable, and to ensure that our future changes in the software did not break our interface with the simulator, then we had to add them to our CI pipeline via the GitHub Actions workflows.

However, we realised again that having them in Python would make it harder for us to extend our tests or create more tests as we progress further in our maintenance cycle, wherefore we needed to refactor it to C#.

C.14 Infrastructure as Code (IaC): Pulumi

We have lately been a bit frustrated at Vagrant. It seems like it lacks some collaborative features that makes it hard to work with. Furthermore, trying to make it collaborate with a configuration management tool, like Ansible, proved to be very challenging and time consuming considering that it should "just be plug and play". We did not make that work, so to eliminate a pain that we currently have, we are looking towards encoding our infrastructure as code now. What is our requirements?

- Should be easy to work with
  - That is, we would prefer to keep the configuration in **YAML** or **C#** as we do not want to increase complexity of learning a new language again
- Can use it in a **GitHub Actions workflow** such that we can create changes in the Minitwit repository, create a pull-request, and then push changes in production when everything is working
- This is not a requirement, but it would be nice if we can use an **open source** tool.
- A tool which can work for multiple cloud providers, such that we can change provider if we want to. However, since we are at Digital Ocean (which we are pretty happy with so far) then we want them to support Digital Ocean.

We looked at the following options:

- Vagrant - Is it still a good option or not?
- Terraform - As stated in a later lecture is something that we are going to look at
- Pulumi - A tool we stumbled upon in a Digital Ocean video
- OpenTofu
- Write code ourselves with a custom client to Digital Ocean

C.14.1 Vagrant

Is a very nice tool for creating infrastructure quickly. What we have experienced so far, however, is that it is hard to collaborate within. It saves metadata files on the local machine to keep some sort of state. It only pushes the creator's `ssh` key to the server, and not the other's `ssh` key, making us copy paste `ssh` keys to the server manually. Last but not least, it is written in a language that none of us have experience with (Ruby). There is not any GitHub Actions for using Vagrant, which we think has something to do with the way that it handles its state. Looking at Vagrant's comparison between Vagrant and Terraform (which is also a tool made by hashicorp) the following is stated:

„Vagrant is a tool for managing development environments and Terraform is a tool for building infrastructure... and more features provided by Vagrant to ease development environment usage... Vagrant is for development environments.[3]"

By this description, Hashicorp has not intended to build Vagrant for the scenario that we use it for today. That is, they do not and will not improve the tool in the direction that we intend to use it. Furthermore, it does not make sense for us to keep Vagrant to setup a local development environment because we already have Docker and Docker Compose that takes care of that.

C.14.2 Terraform

As we can see in the above section, Terraform is a product that is used for the specific functionality that we need. It can integrate with our GitHub workflow. It does so with Terraform Cloud to keep the state of the infrastructure. It can create infrastructure on multiple cloud vendors. It also has support for Digital Ocean.

One of the downsides is that they introduce their own language: HCL. This is something that we do not want to have. We want to limit the amount of languages that our repository contains to a minimum. Although we can see that they actually have what they call a CDK (Cloud Development Kit) for Terraform, however, we doubt that is their first priority to maintain. An upside to the CDK is that it can add testing to Terraform.

Terraform is not open source, which makes it harder to trust the tool. A quick search on the internet yields that others have forked a former repository of Terraform and created a Terraform look-a-like, called OpenTofu, which reveals that the community does not trust Hashicorp's development of the tool, now that they have transitioned from open source to a Business Source License.

C.14.3 Pulumi

Pulumi is similar to Terraform, in the way that it also provides a cloud solution to keep state of the infrastructure. It does not offer a new language to write the infrastructure, but instead offers SDKs in languages that we already know. Both Pulumi and Terraform follow what is called

Desired State Infrastructure, meaning that they try to keep the infrastructure in the desired state that the code describes.

It contains support for CI / CD tools, so we can integrate it into our existing workflows. In contrast to Terraform, it is open source. Pulumi seems to be widely adopted and is known for being easy for developers to get started.

C.14.4 OpenTofu

OpenTofu is a forked version of Terraform. It keeps the structure in HCL. What makes this interesting is that the community has actively taken a decision to maintain this project in order for it to not be a BSL tool, distancing them from Hashicorp's decision with Terraform. This can be seen in the OpenTofu Manifesto[6]. Another advantage to OpenTofu is that it has later been adopted in the Linux Foundation and for sure is here to stay.

As it is a copy of Terraform it has some of the same features and downsides. This can be seen by having their own custom language to describe the infrastructure called OpenTofu Language. OpenTofu has not done the same thing as Terraform in terms of creating a CDK. That means the only option is the OpenTofu Language.

Another downside is the fact that we are unsure whether there is support for Digital Ocean as a provider. We would think that there would be, but it is not clear and we will need to spend time looking into it.

C.14.5 Write C# code with Digital Ocean client

This option may not be the best option. Compared to the others we will have a tight coupling to Digital Ocean. This will make it challenging for us to change provider in the future, if that is needed. This will also require us to keep an eye out for the changes to the general API and make adjustments to the custom client that we would have to make. We could also find a client that is made for us, written in C#, but then we would have to find one that is being maintained.

We have realised that the maintenance of just written code for the infrastructure, becomes a project in itself. We would have to manage an abundance of other dependencies, which is taken care of from the other projects. Also we do not have any testing capabilities as the other tools have.

Furthermore, we will not benefit from things that others may have created. A large community for an open source tool allows for sharing solutions and code for that specific tool. We do not really have that option here, as we are going to create a complete new tool with this option.

C.15 Scaling: Docker Swarm

We need to add horizontal scaling to our system.

Our requirements

- Open source
- Works well with Ansible/Docker/Pulumi.

C.15.1 Docker Swarm

[1]

- Open source
- It is based on the Docker API, and therefore is **lightweight**, and **easy to use**.
- Docker Swarm allows for limited scaling, and is not recommended for big systems, so if Minitwit ever becomes a bigger service, this could potentially be a problem.
- It is easy to add new nodes to existing clusters as both manager and worker
- Compared to other tools like kubernetes
- Simple to install compared to Kubernetes
- Same common language that we are already using to navigate in the structure
- Limited guides on interactions between Docker Swarm and Ansible.

C.15.2 Building our own redundant load balancer

- Time-consuming to create
- Less information on how to build it
- Probably more challenging to add new nodes than for Docker Swarm

C.15.3 Kubernetes

- Takes more planning to implement than Docker Swarm
- Has a quite steep learning curve
- It can sustain and manage large architectures and complex workloads
- Has a GUI

C.16 Upgrade/Update Strategy: Rolling updates

Rolling updates is the default update strategy in Docker Swarm and therefore the one we have chosen so as to not make the implementation more complicated and “home-made” than it needs to be.

C.17 Precommit

D Log

E Security Assessment

E.1 Risk Identification

E.1.1 Asset Identification

By browsing through our setup and documentation, we identified the following list of assets:

- Web application
- Public GitHub repository
- Digital Ocean servers
- Tools
  - Ansible
  - Code Climate
  - Docker, Docker Compose & Docker Swarm
  - GitHub Actions
  - Grafana
  - Linters
  - Loki
  - Pulumi
  - SonarCloud

E.1.2 Threat Source Identification

To help us identify possible threats to the system, we have consulted the OWASP *Top 10 Web Application Security Risks*[7], which describes the following threats:

1. Broken Access Control
2. Cryptographic Failures
3. Injection Attacks
4. Insecure Design
5. Security Misconfiguration
6. Vulnerable and Outdated Components
7. Identification and Authentication Failures
8. Software and Data Integrity Failures

9. Security Logging and Monitoring Failures
10. Server Side Request Forgery

E.1.3 Risk Scenario Construction

Based on the information gathered from the two previous steps, we have constructed the following risk scenarios and outlined which of the OWASP top 10 risks that would affect the described scenarios:

1. **URL Tampering**

   The attacker can construct a URL in the **web application** with a user ID, such that they can bypass login and are able to write a message from another user's account.

   This would be an issue of *Broken Access Control* and *Server Side Request Forgery*.

2. **Log Injection**

   The attacker can fabricate log information via an injection attack in the **web application** as a means to hide their activity and ill-intentioned actions.

   This would be an issue of *Security Logging and Monitoring Failures* and *Injection Attacks*.

3. **Password Brute Forcing**

   The attacker can brute force login credentials in the **web application** by taking advantage of no timeouts and a weaker hash implementation.

   This would be an issue of *Cryptographic Failures* and *Identification and Authentication Failures*.

4. **Depricated Dependencies**

   The attacker can identify weak, outdated or depricated tools and dependencies in the system's CI/CD pipeline via **GitHub Actions**, which is publicly availble through the **GitHub repository**.

   This would be an issue of *Vulnerable and Outdated Components* and *Software and Data Integrity Failures*.

5. **Open Ports**

   The attacker can scan the public IP addresses of the **Digital Ocean** servers to find unnecessarily or unexpectedly open ports with known vulnerabilities, which can be exploited in further attacks.

   This would be an issue of *Security Misconfiguration*.

6. **SQL Injection**

   The attacker can target the login page of the **web application** with SQL-injection attacks to strike the database.

   This would be an issue of *Injection Attacks*.

7. **Exposed Secrets**

   The attacker can get access to the system, or associated tools, via secrets found written in the code in the public **GitHub repository**.

   This would be an issue of *Identification and Authentication Failures* and *Security Misconfiguration*.

E.2 Risk Analysis

E.2.1 Likelihood Analysis

Likelihood will be graded on the following scale: {Rare, Unlikely, Possible, Likely, Certain} with *Rare* being the least severe and *Certain* being the most severe.

1. **URL Tampering**

   We examined the different URLs on the web application and could not find any where the IDs or login parameters were exposed. Therefore we deemed the likelihood to be **Unlikely**.

2. **Log Injection**

   We received warnings from SonarCloud that we had logging vulnerabilities in our code several different places. Therefore we deemed the likelihood to be **Likely**.

3. **Password Brute Forcing**

   We currently do not have any measures in place to combat brute force attacks and no password requirements for the users when signing up. Therefore we deemed the likelihood to be **Possible**.

4. **Depricated Dependencies**

   Our GitHub repository, and thereby our workflows for GitHub Actions as well, are public for anyone to see. In the workflows we are using templates of actions made available and written by others, alongside showcasing some of various tools we use. We currently rely on the fact that the actions and tools we use are secure, but have not incoporated anything that checks whether that is true. However, many of the tools we use are well-known and therefore we hope that known vulnerabilities are getting discovered rather quickly. Therefore we deemed the likelihood to be **Possible**.

5. **Open Ports**

   All of the IP addresses for our servers are public in Digital Ocean, which makes it very easy to scan for vulnerabilities. We have put up firewalls and have taken measures to only have necessary ports open, but are also aware that accidental port exposure, through some of the tools we are using, is possible. We are convinced that this is how our server got hacked during the course. Therefore we deemed the likelihood to be **Certain**.

6. **SQL Injection**

   We have made sure to sanitize user input on the login page of the web application. Therefore we deemed the likelihood to be **Rare**.

7. **Exposed Secrets**

   We have been conscious to ensure that secrets are either kept locally where only ourselves can access them, such as secret keys for logging into the servers, or used the "environment secrets" tool on GitHub, if secrets had to be accessed from the repository. Additionally, we have not made generic passwords, but rather used random password generators to get stronger passwords. Therefore we deemed the likelihood to be **Rare**.

E.2.2 Impact Analysis

Impact will be graded on the following scale: {Insignificant, Negligible, Marginal, Critical, Catastrophic} with *Insignificant* being the least severe and *Catastrophic* being the most severe.

1. **URL Tampering**

   This would breach both the confidentiality and the integrity of the system's security. We still keep all our data, though the data would have been compromised. Therefore we deemed the impact to be **Critical**.

2. **Log Injection**

   This could be used to disguise an attacker's activity and attack attempts on the web application. However, it would not give them access to the server or applicition itself, nor other data than the logs. We would still want to know if someone was trying to attack our system. Therefore we deemed the impact to be **Marginal**.

3. **Password Brute Forcing**

   This would breach both confidentiality and integrity. In very severe cases, it could also affect the availability, if the requests to login became too intense. We would still keep all of our data, though the data would have been compromised. Therefore we deemed the impact to be **Critical**.

4. **Depricated Dependencies**

   This would have a very big attack surface, as we most likely would not know which tool or where in the application process we could have a vulnerability. The target for a vulnerability could thereby vary in severity, but could in the worst case have severe consequences. Therefore we deemed the impact to be **Catastrophic**.

5. **Open Ports**

   This again would have a big attack surface, as we do not know which tool or where in the application we potentially could have a vulnerability exposing ports for an adversary to attack. If the attacker ended up gaining access to our servers, they would have full control over the application in the worst case. Therefore we deemed the impact to be **Catastrophic**.

6. **SQL Injection**

   This would breach both confidentiality and integrity. The data could both be compromised and lost, but we would have a backup of the database. Therefore we deemed the impact to be **Critical**.

7. **Exposed Secrets**

   If any of the secret in the GitHub repository were to fall into an attacker's hands, they would be able to get access to our setup and thereby dismantle the entire system. Therefore we deemed the impact to be **Catastrophic**.

E.2.3 Risk Matrix

Based on the points made about the likelihood and impact of each risk scenario, we have constructed the following risk matrix to indicate the severities and prioritize the scenarios:

	Rare	**Unlikely**	**Possible**	**Likely**	**Certain**
Catastrophic	Exposed Secrets	🟥	Depricated Dependencies	🟥	Open Ports
Critical	SQL Injection	URL tampering	Password Brute Forcing	🟥	🟥
Marginal	🟩	🟩	🟨	Log Injection	🟥
Negligible	🟩	🟩	🟩	🟨	🟥
Insignificant	🟩	🟩	🟩	🟩	🟨

Figure 11: *Risk matrix based on security assessment*

E.2.4 Action Plan

We discussed the results of the security assessment and decided on focusing on the risk scenarios placed in the red area of the risk matrix.

For the open ports, we went through our firewall settings, which were handled by the Ansible ufw module. Here we scanned our main server's IP address with *nmap* to see the current open ports. We tried to close the port we had open for Prometheus, but got conflicting results when we checked the firewall itself compared to the scan of the IP address. The cause of this is expected to be Docker bypassing the firewall through IP tables.

For the log injection, we went through our code and ensured that all user data was sanitized before added to the log, such that it could not be tampered with.

For the depricated dependencies, we added the tool *dependabot* to our CI/CD pipeline, which makes sure that the dependencies used in our Minitwit system are up to date and thereby less vulnerable to older known exploits.

For the password brute forcing, we decided to leave it as it, due to the group not having enough information on how the creation of users and login works in the simulator and thereby not knowing if the simulator could handle password restraints or 2-factor authentication, which could have been our solution to improve this issue.